Reinforcement Learning: An Introduction
Solutions: Chapter 6

Mrinank Sharma

April 7, 2020

Exercise 6.1: TD to MC error

We have:

$$\delta_t = R_{t+1} + \gamma V_t(S_{t+1}) - V_t(S_t), \tag{1}$$

i.e., the error using the estimates at time t. We can write the update rule:

$$\begin{aligned} V_{t+1}(S_t) &= V_t(S_t) + \alpha[R_{t+1} + \gamma V_t(S_{t+1}) - V_t(S_t)] \\ &= V_t(S_t) + \alpha\delta_t. \end{aligned} \tag{2}$$

Rearranging yields:

$$V_{t+1}(s) - V_t(s) = \begin{cases} \alpha\delta_t, & s = S_t \\ 0, & \text{otherwise} \end{cases}. \tag{3}$$

Now, we follow the same steps as in the book.

$$\begin{aligned} G_t - V_t(S_t) &= R_{t+1} + \gamma G_{t+1} - V_t(S_t) \\ &= R_{t+1} + \gamma V_t(S_{t+1}) - V_t(S_t) + \gamma[G_{t+1} - V_{t+1}(S_{t+1}) + V_{t+1}(S_{t+1}) - V_t(S_{t+1})] \\ &= \delta_t + \gamma\alpha\mathbb{I}\{S_{t+1} = S_t\} + \gamma[G_{t+1} - V_{t+1}(S_{t+1})] \\ &\vdots \\ &= \sum_{k=t}^{T-1} \gamma^{k-t}\delta_k[1 + \alpha\gamma\mathbb{I}\{S_{k+1} = S_k\}]. \end{aligned} \tag{4}$$

This is incredibly similar as before, but we have additional $\alpha\gamma$ terms if the new state was the same as the old one, because only in these do we need our correction term.

Exercise 6.2: MC vs TD

Let's use the driving example where we have moved office, and let's suppose that we have learnt the correct V for the previous office. The MC estimate would start again, taking time to converge to the correct V and not utilise previous information that we have i.e., the old estimates of V (it won't bootstrap).

Note that V is the *expected time to go* from a state.

TD methods effectively from each step will compare the length of time taken at each step to how long it was previously expected to take. The first passthrough will update the value of each state to account for the difference in time taken to get to the next state. Future TD passthroughs will propagate this information backwards. TD is helpful because the expected future cost to go is the same for future steps in the trajectory, so we should use this. Monte Carlo doesn't utilise this. MC doens't really use the MDP structure.

Exercise 6.3: TD Example

The first episode ended in hitting left of A `A`. Mostly no updates since all values are initialised to the same. When it hits the left hand side,

$$V(\texttt{A})_1 = V(\texttt{A})_0 + \alpha[R_t + V(\texttt{terminal}) - V(\texttt{A})_0]. \tag{5}$$

The update is then:

$$\alpha[0 + 0 - V(\texttt{A})_0] = 0.05. \tag{6}$$

with $\alpha = 0.1$ and the initialisation used. This matchest the graph.

Exercise 6.4: Changing α

Note that for the MC results, constant α is used. I suggest that the algorithm might perform better using a sample average since the problem is stationary, so weighting all results the same should perform well. This isn't a fixed value though.

Increasing α makes the curve less smooth and vice versa. We get a good range in behaviour for MC; the smallest α is incredibly smooth whilst the largest α fluctatues a bit more, so the alphas suggested seem reasonable. A wider range would only accentuate this.

We only see 3 choices of α for TD. It seems plausible to me that a smaller value of α could perform better (as it would be smoother) after 100 episodes, but I'm not sure. A larger value of α than suggested would drop faster but also fluctuate a lot.

I'd say that we've seen enough values of α to make our conclusion and see the patterns.

Exercise 6.5: Strange TD Behaviour

Given our initialisation, the largest components of the RMS error will be due to states `A` and `E`. The error from the central states will be smaller. In general, we'd expect our TD updates to be pretty non-smooth with larger values of α. Additionally, the form of the updates will first affect the edge values which will then move backwards to the centre.

I suggest that what is happening is that first, the values of the edge states get pulled far closer to their true values. At these points, all of the values are reasonably close to their true values. As the updates continue, the stochasticity affects the inner states more, resulting in the error increasing.

This is only a guess! But, if this is true, the initialisation is important and this behaviour wouldn't be seen for all values.

Exercise 6.6: Computing Values

We could have either:

1. Solved the linear system of equations to compute values.
2. Performed Iterative DP policy evaluation.

The linear system of equations is easy to solve in this case, so I suspect that is what was used.

Exercise 6.7: Off-policy TD(0)

The TD(0) update uses that:

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s], \tag{7}$$

converting this into an update rule. Following the standard IS derivation and converting that into an update rule, we compute the off-policy TD(0) update:

$$V_\pi(S_t) \leftarrow V_\pi(S_t) + \alpha[\rho_{t:t}(r + \gamma V_\pi(S_{t+1})) - V_\pi(S_t)], \tag{8}$$

where $\rho_{t:t}$ is defined in the usual way.

Exercise 6.8: Q MC and TD Errors

The MC error is:

$$\begin{aligned} G_t - Q(S_t, A_t) &= R_{t+1} + \gamma G_{t+1} - Q(S_t, A_t) \\ &= \delta_t + \gamma(G_{t+1} - Q(S_{t+1}, A_{t+1})) \\ &= \sum_{k=t}^{T-1} \gamma^{k-t} \delta_t, \end{aligned} \tag{9}$$

which is precisely the form required, assuming that the action-value function doesn't change.

Exercise 6.9 & 6.10: Windy Gridworld

Figures 1, 2 and 3 show the results for this programming exercise.

As expected, introducing the possibility for diagonal moves is a big advantage. The noise in the wind also makes a large difference.

Exercise 6.11: Q-Learning

Q-Learning is an off-policy methods because it learns the action-value function for the *optimal policy* whilst acting with a sub-optimal, behavioural policy.

Exercise 6.12: Sarsa vs Q-Learning

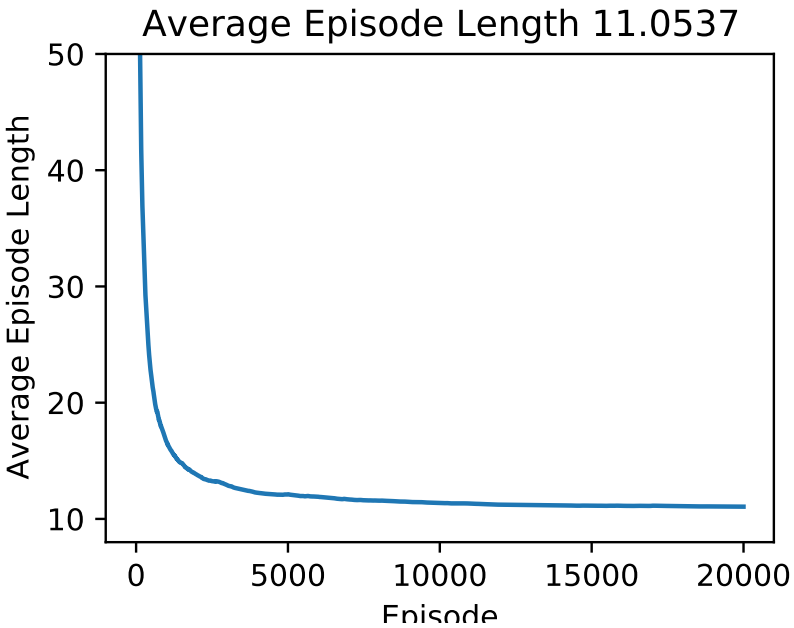

Figure 1: Windy gridworld results with added diagonal moves.

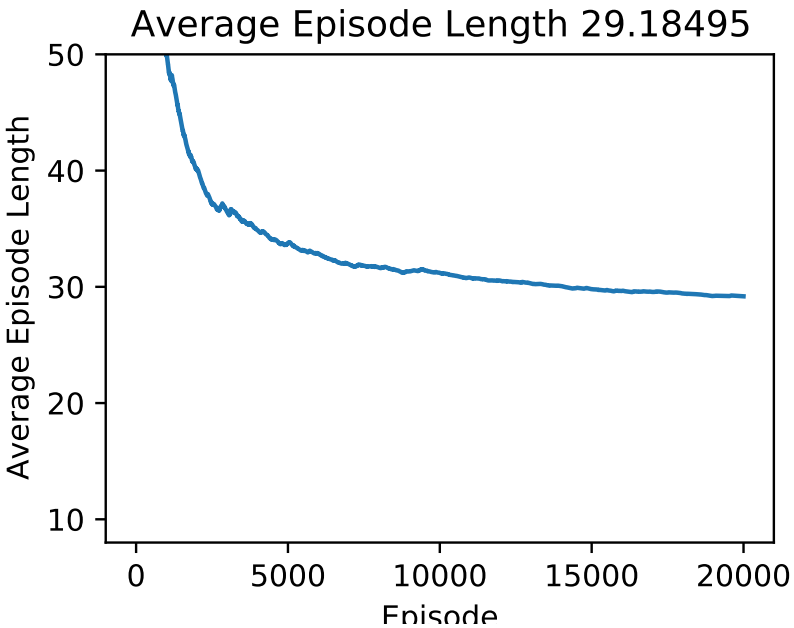

Figure 2: Windy gridworld results with added noise in the wind.

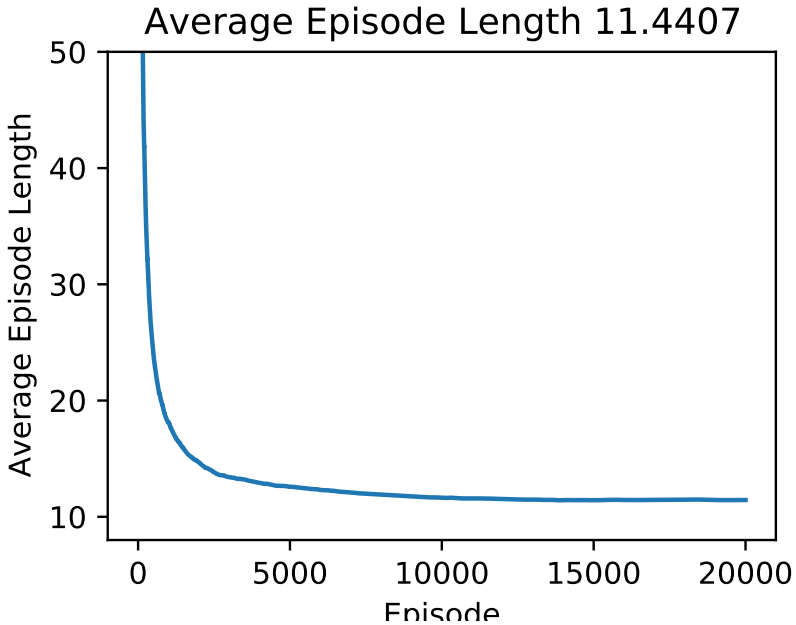

Figure 3: Windy gridworld, no noise but allow for staying still.

Suppose **all** action selection, across both Q-Learning and Sarsa is greedy. Then Sarsa picks the optimal action to perform in a state S, performs this, observes state S'. It then samples the optimal action for S', and uses this Q value for bootstrapping.

Instead consider Q-Learning. All action selection is greedy, so given state S, we pick the same optimal action as in Sarsa. Similarly, we perform this action, observe the new state and use the optimal action in this state for bootstrapping.

So yes, in this case, both algorithms are the same. They are however both useless, as they will only update the visited states.

Exercise 6.13: Double Expected Sarsa

The general algorithm is as follows.

Algorithm 1 Double Expected-Sarsa

Parameters:

step size $\alpha \in (0,1]$

small $\epsilon > 0$

Initialisation:

$Q_1(s,a)$ arbitarily for all $s \in \mathcal{S}, a \in \mathcal{A}$ except that $Q_1(\texttt{terminal},\cdot) = 0$

$Q_2(s,a)$ arbitarily for all $s \in \mathcal{S}, a \in \mathcal{A}$ except that $Q_2(\texttt{terminal},\cdot) = 0$

loop for each episode:
  initialise S
  loop for each episode step:
    Choose A using $Q_1(S,\cdot) + Q_2(S,\cdot)$ e.g., ϵ-greedy. ▷ Off-Policy Method; this is the behavioural policy
    Take action A, observe S', R
    if `coin-toss` is `heads` **then**
      derive policy π_1 from Q_1 ▷ e.g. greedy policy yields Q-Learning, ϵ-greedy
      $Q_2(S,A) \leftarrow Q_2(S,A) + \alpha[R + \gamma \sum_a \pi_1(a,S')\, Q_2(S',a) - Q_2(S,A)]$
    else
      derive policy π_2 from Q_2
      $Q_1(S,A) \leftarrow Q_1(S,A) + \alpha[R + \gamma \sum_a \pi_2(a,S')\, Q_1(S',a) - Q_1(S,A)]$
    $S \leftarrow S'$

For the specific question, simply substitute in the target epislon greedy policy values i.e.,

$$\pi_1(a|S') = \begin{cases} \epsilon/|\mathcal{A}(S')|, & a \neq \arg\max Q_1(S',a) \\ 1 - \epsilon + \epsilon/|\mathcal{A}(S')|, & \text{otherwise} \end{cases}, \tag{10}$$

and similarly for π_2.

Exercise 6.14: Afterstates

In Jack's policy rental, the initial part of the dynamics is known fully i.e., how many cars will move. What is not known is how many cars will arrive or be requested. We can formulate this directly using *afterstates*, where the afterstate is the number of cars at each location *after* cars have been moved from one location to the next.

This is expected to speed up convergence, for the general reason that we expect afterstates to be useful. There are many state-action pairs which lead to the same afterstate, and thus should have the same value. For example, $(10, 10)$ and moving 2 cars to give $(8, 12)$ ought to have the same value as being in state $(9, 11)$ and only moving 1 car. Afterstates add a helpful inductive bias, utilising additional knowledge of the problem, and thus they ought to be helpful.